UNIVERSITY OF PLYMOUTH
School of Engineering, Computing & Mathematics

# Lecture 6: Client-side & Sever-side Testing

## Full-Stack Development

Mark Dixon

School of Engineering, Computing and Mathematics

# Last Session

- What did we do last session (write down topics, what you can remember)?
  - Did you learn anything (was anything particularly useful or good)?
  - How did the lab session go (were you able to get something running)?
  - What could be better?

# Introduction

**Today's topics**

1. Unit testing in JavaScript
2. Server-Side Testing
3. Server-Side Unit Testing
4. Mock Objects
5. Integration Testing
6. Website accessibility (and evaluating it)
7. Usability testing

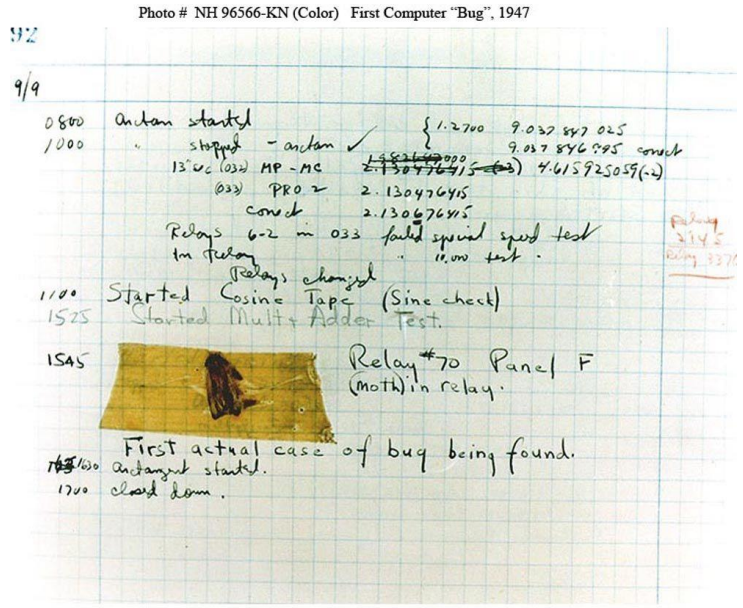
Session learning outcomes – **by the end of today's lecture you will be able to:**

- Write simple unit tests to exercise JavaScript code in the browser
- Implement unit tests for server-side JavaScript code
- Use mock objects to represent other parts of the system within tests
- Test the connections between components of a system using integration tests
- Assess the accessibility of a web page
- Conduct a usability study to test the user-acceptance of your software

# Why test software?

**"I'll finish coding and test if I have time"**

- No! Testing is an integral part of the development process
- Can actually **save** you time in the long run
- **Reduces bugs** and **improves the quality** of software



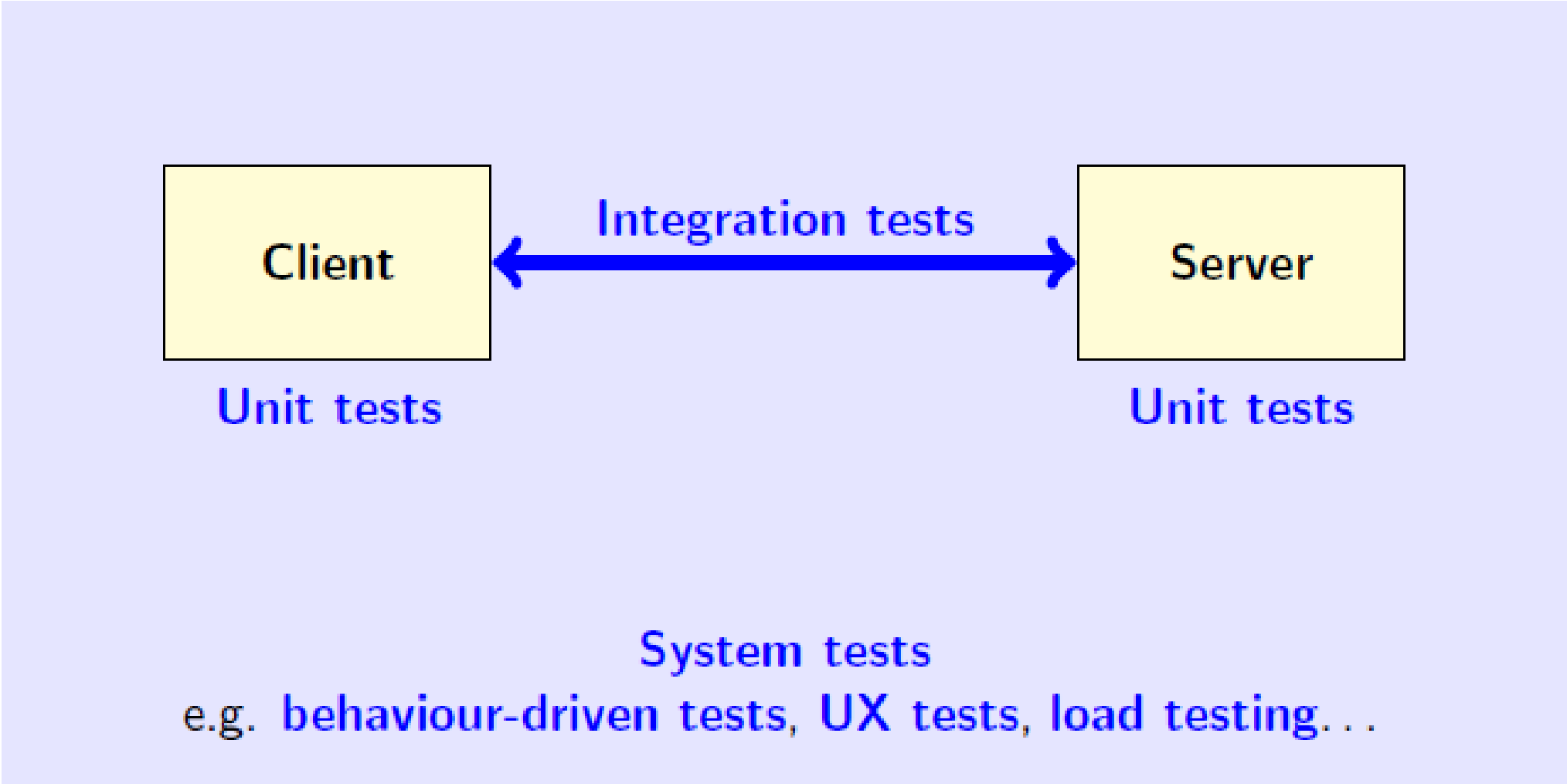

## Testing considerations

- Test case design – consider **edge cases**, **corner cases**…
- Code coverage
- **Black box** vs **white box** testing

# Testing - Types

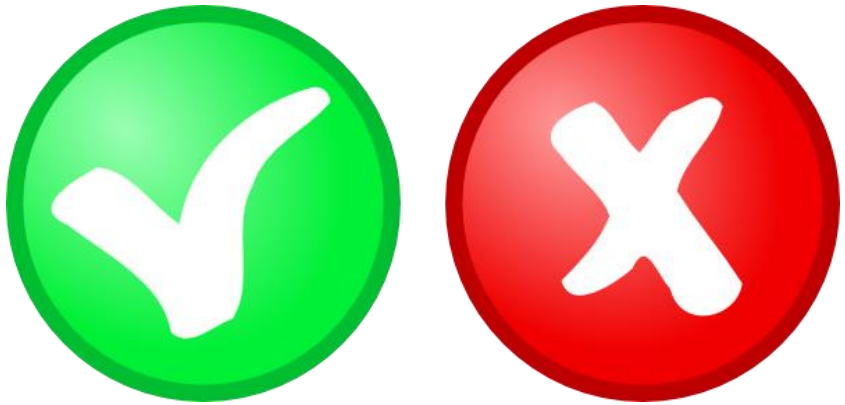
- Unit Testing – checks individual functions/procedures within a file
- Integration Testing – checks interaction between components
- System / End-to-end Testing – checks operation of whole system
- User Testing – System may match the specification / design, but is it:
  - Useful
  - Usable
  - Learnable
- Smoke Test – checks a few critical things (quickly)
- Penetration Testing
- Performance Testing
- …

# Full-stack testing

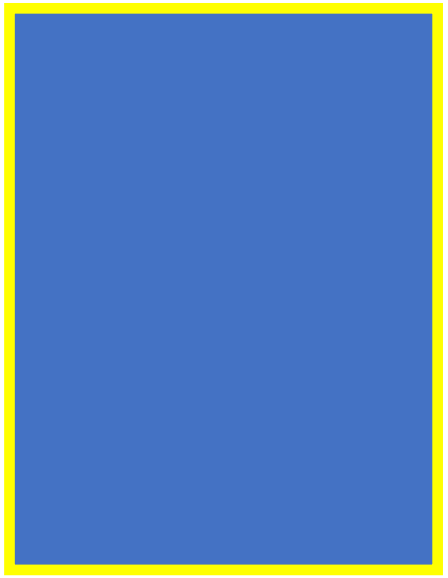


# Unit testing

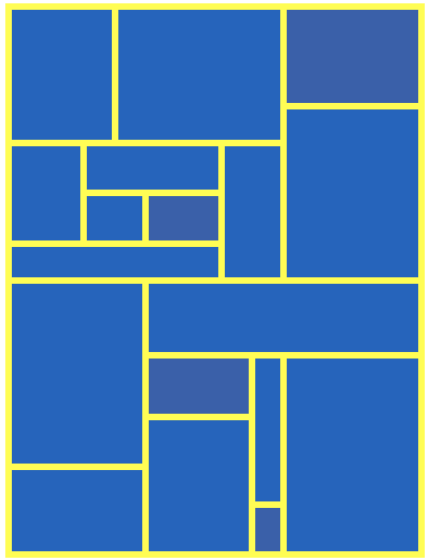
- Testing reduces the number of bugs
- Automated tests can run every time we change the code
- Automated tests can run before we deploy (and stop deploy on fail)
- Supports regression testing
  - When we change old code, old tests ensure that we don’t introduce new bugs
- Tests also provide documentation of what the code does
- Unit testing is a framework for providing **tests as code**



# Unit Testing



- Traditional Testing
- whole system tested
- Errors:
  - Easily undetected
  - difficult to track down



- Unit Testing
- Each part of system tested individually
- Test scope much smaller, errors
  - detected earlier
  - isolated

# Unit Testing: What is a Unit?

## How can we test this:

```
let express = require("express");
let app = express();

app.get("/sayHello", function(request, response) {
    response.send("Hello World, from Express");
});

app.listen(9000);
```

## How can we test this:

```
let express = require("express");
let app = express();

function sayHello(request, response) {
    response.send("Hello World, from Express");
}

app.get("/sayHello", sayHello);
app.listen(9000);
```

# Unit tests

## What is a test?

- A test executes a unit of code and checks whether the result is as expected
  - Actual / Observed value
  - Expected value
  - Assertion (equal, OK…) and feedback message

## Verify that a sum function works

```
function sum(a, b) {
    return a + b;
}
```

## A simple unit test

```
test("Test the sum function", function() {
    chai.assert.equal(sum(5, 12), 17, "5+12 should equal 17");
});
```

# Unit Tests: AAA

Arrange, Act, Assert (AAA) approach

- Determine data needed (manual testing)
  - try running with really simple data / "dummy data“
  - can help quickly assess if unit works well
- **Arrange**: set up the unit to be tested
  - bring system to desired state & configure (internal) dependencies
- **Act**: call unit (function / procedure) to be tested
  - pass dependencies & capture output value (if any)
- **Assert**: check that actual outcome matches expected
- Analyse Test results
  - Failure(s) trigger debugging (cause and solution)

# Independent tests

- Unit tests should be **atomic**
- They should not rely on side-affects of other tests
- The order in which they are executed should not matter

**To make a test atomic**

- Get original values from the application/web page
- Set up the values for the test
- Restore the original values after the test has run

**Don’t**

- Store values in a variable that other tests might change (e.g. a click counter)

# Mocha and Chai

## Mocha

- A JavaScript unit testing framework
- Runs in Browser (client-side) & Node (server-side)
- https://mochajs.org/


## Chai

- Assertion library for inclusion in JS unit tests
- Runs in Browser (client-side) & Node (server-side)
- https://www.chaijs.com/

# Client-side Testing: Writing a unit test

1. Set up the test function stub
2. Preserve any original page properties
3. Set up the test conditions
4. Test the result of updating the conditions
5. Reset the page properties

```
suite("Test suite description", function() {
    test("Test 1 description", function() {
        // Test code goes here.
    });

    test("Test 2 description", function() {
        // Test code goes here.
    });
});
```

# Client-side Testing: Writing a unit test

## Preserve the original page properties

```
let originalCol = $(".light").css("background-color");
```

## Set up the test conditions

```
$(".light").css("background-color", "#00ffff");
$("#button").trigger("click");
```

## Test the result of updating the conditions

```
let lightCol = rgb2hex($(".light").css("background-color"));
chai.assert.equal(lightCol, "#ffff00", "Class has wrong colour");
```

## Reset the page properties

```
$(".light").css("background-color", originalCol);
```

# Client-side Testing: Hooks

Suites allow repeated code to be placed in a single place

## Available hooks

- **suiteSetup** – runs before the **first** test
- **suiteTeardown** – runs after the **last** test
- **setup** – runs before **each** test
- **teardown** – runs after **each** test

```
suite("A suite", function() {
    suiteSetup(function() {
        // Prepare something once for all
        // tests
    });
    suiteTeardown(function() {
        // Clean up once after all tests.
    });
    setup(function() {
        // Prepare something before each
        // test.
    });
    teardown(function() {
        // Clean up after each test.
    });
    test("A test", function {
        // Test code here…
    });
});
```

# Test-driven development

1. **Write the test**

2. **Watch it fail**
   - Confirm that it passes when it should
   - Find bugs in the test code
   - Run all tests alongside a new one

3. **Make it pass**
   - Simplest possible implementation that causes it to pass

4. **Refactor the code**

# Client-side Testing: “Headless” testing

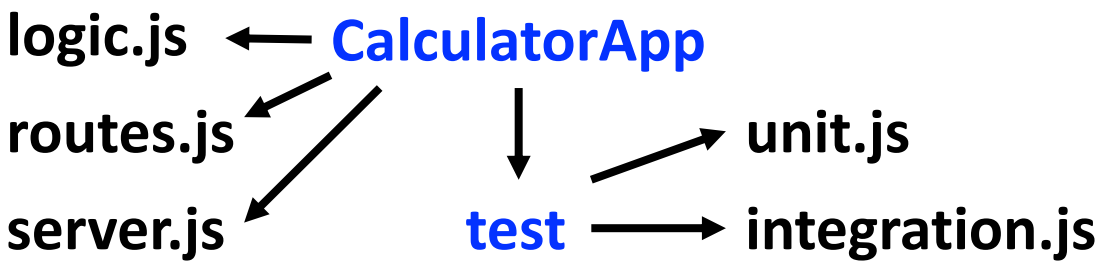
## Browser testing without a browser

- **PhantomJS** renders the page without starting a browser (can use JS to interact with the page and read out effects)
- **Headless Chrome** accesses and renders a page without opening the browser window – can use **Puppeteer** to write tests that interact with the page

```
const puppeteer = require(“puppeteer”);

(asynch() => {}
    const browser = await puppeteer.launch();
    const page = await browser.newPage();
    await page.goto("https://example.com");
    await page.screenshot({path: “example.png”});
    browser.close();
})();
```

# Server-side Testing: A sample full-stack application




```
function square(x) {
    return x * x;
}
```

logic.js

routes.js

```
let logic = require("./logic");

function square(req, res) {
   let num = req.params.number;

   res.send(logic.square(num)
                    .toString());
}

module.exports.square = square;
```

```
let express = require("express");
let routes = require("./routes");

let app = express();

app.get("/square/:number", routes.square);

module.exports.app = app;
```

server.js

# Server-side Testing: Testing the square function

```
let chai = require("chai");
let logic = require("./logic");

suite("Test square function", function() {

    test("Test the square function", function() {
        let result = logic.square(3);
        chai.assert.isNumber(result, "Result should be numeric");
        chai.assert.equal(result, 9, "3x3 should equal 9");

        result = logic.square(5);
        chai.assert.isNumber(result, "Result should be numeric");
        chai.assert.equal(result, 25, "5x5 should equal 25");

    }

}
```

# Server-side Testing: Running the tests

## Use the mocha application to run the tests

- The **mocha** application has a range of test interfaces – specify the TDD one
- Tests are stored in a directory called **test**
- Mocha will execute all of the tests within the directory

```
Davids-MacBook-Pro:CalculatorApp djw213$ mocha -ui tdd test/


  Test square function
    ✓ Test the square function


  1 passing (11ms)
```

# Server-side Testing: Overview

- To run server:

  **`node server.js`**

- To run tests:

  **`mocha -ui tdd test/`**

```
Test sayHello
  ✓ Test sayHello


1 passing (4ms)
```

***functions.js***

```
function sayHello(){
  return "Hello world!!";
}

module.exports.sayHello = sayHello;
```

***test/unit-tests.js***

```
let chai = require("chai");
let functions = require("../functions");

suite("Test sayHello", function() {

  test("Test sayHello", function(){
    let expected = "Hello world!!";          // Arrange.
    let actual = functions.sayHello();       // Act.
    chai.assert.equal(expected, actual);     // Assert.
  })
})
```

***server.js***

```
let express = require("express");
let functions = require("./functions");
let app = express();
let port = 9000;

app.get("/hello", function(request, response){
  response.send(functions.sayHello());
})

app.server = app.listen(port, function () {
  console.log("Listening on " + port);
});
```

# Server-side Testing: Mock objects

Use mock objects when the real object

- Has non-deterministic behaviour
- Is difficult to set up
- Has behaviour that is hard to trigger (e.g. network error)
- Is slow
- Has (or is) a user interface
- Does not yet exist

**Implementing mock objects**

1. Use an interface to describe the object
2. Implement the interface for production code
3. Implement the interface in a mock object for the test

# Server-side Testing: Sinon example - timers

## Control the system time for time/date specific tests

1. Initialise the date/time as required
2. Run the test
3. Restore the actual date/time

```
let chai = require("chai");
let sinon = require("sinon");
let logic = require("./logic");

suite("Test message generator", function() {
    test("Check morning message is correct", function() {
        let date = new Date(2021, 11, 1, 10, 0, 0, 0);
        let clock = sinon.useFakeTimers(date);
        let msg = logic.greetingMessage();
        chai.assert.equal("Good morning", msg, "Wrong message for 10am");
        clock.restore();
    }
}
```

# Server-side Testing: Sinon example - timers

```
test("Check morning message is correct", function() {
    let date = new Date(2021, 11, 1, 14, 0, 0, 0);
    let clock = sinon.useFakeTimers(date);
    let msg = logic.greetingMessage();
    chai.assert.equal("Good afternoon", msg, "Wrong message for 2pm");
    clock.restore();
}

test("Check morning message is correct", function() {
    let date = new Date(2021, 11, 1, 21, 0, 0, 0);
    let clock = sinon.useFakeTimers(date);
    let msg = logic.greetingMessage();
    chai.assert.equal("Good evening", msg, "Wrong message for 9pm");
    clock.restore();
}
```

```
Davids-MacBook-Pro:HelloWorld djw213$ mocha -ui tdd test


  Test message generator
    ✓ Check morning message correct
    ✓ Check afternoon message correct
    ✓ Check evening message correct
```

# Server-side Testing: Sinon example - spies

## Inspect the calling of functions

- e.g. has a function been called?

```
Davids-MacBook-Pro:HelloWorld djw213$ mocha -ui tdd test


  Test Express router
    ✓ GET greetingRoute
```

```
let chai = require("chai");
let sinon = require("sinon");
let routes = require("./routes");

suite("Test Express Router", function() {
    test("GET greetingRoute", function() {
        let request = {};
        let response = {};
        response.send = sinon.spy();

        routes.greetingRoute(request, response);
        chai.assert.isTrue(response.send.calledOnce);
    }
}
```

# Possible integration failures

## Possible integration failures

- Incorrect method invocation
- Methods invoked correctly but in the wrong sequence
- Timing failures – **race condition**
- Throughput/capacity problems

**To test these, your integration tests will…**

1. Read/write to a database
2. Call a web service
3. Interact with the file system

# Testing the square route

```
Davids-MacBook-Pro:CalculatorApp djw213$ mocha -ui tdd test/


  Test routes
    ✓ Test GET /square

  Test square function
    ✓ Test the square function


  2 passing (77ms)
```

```
let chai = require("chai");
let chaiHttp = require("chai-http");
let server = require("../server");

chai.use("chaiHttp");

suite("Test routes", function() {
    test("Test GET /square", function() {
        let app = server.app;

        chai.request(app).get("/square/3")
            .end(function(error, response) {
                chai.assert.equal(response.status, 200, "Wrong status code");
                chai.assert.equal(response.text, "9", "Wrong response text");
            )};
    });
});
```

# Integration testing a database

**Set up a test database to run integration tests against it**

1. **Before any of the tests run** set up a test instance of the database
2. **Before each test** set up test data so that tests are executed against a standard setup
3. Execute the test
4. **After each test** clean the data from the database
5. **At the end of the suite** delete the test database

# Best practice

1. Integrate early, integrate often
2. Don’t test business logic with integration tests
3. Keep test suites separate
4. Log often
5. Follow a test plan
6. Automate whatever you can

# User Testing: Web page accessibility

## What is “accessibility”?

- It should be possible for all people to perceive, understand and interact with the web
- And contribute to the web
- Many disabilities affect a person’s ability to do so
- Also important to include people without disabilities – e.g., older people, different devices, poor internet connections…

# User Testing: Web page accessibility

**Why is it important?**

- Legal considerations – e.g., public sector websites have accessibility **requirements**
- Maximise opportunities for all users to engage with your app

**How to maximise accessibility?**

- Support screen readers (e.g., `alt` attributes, form elements with associated **labels**)
- Visual considerations such as **contrast**
- Maximise use of **screen readers**

More information at https://www.gov.uk/guidance/accessibility-requirements-for-public-sector-websites-and-apps.

# User Testing: new HTML5 (semantic & non-semantic)

HTML5 specification contains elements that are used to divide up a page into more meaningful elements – this is helpful for **screen readers**

| | |
|---|---|
| `<section>` | a generic section of a document – groups content by themes like the sections of a report |
| `<article>` | a self-contained item of content, such as a blog post |
| `<nav>` | used to encapsulate large navigation blocks |
| `<aside>` | similar to a sidebar in a newspaper, could be used for advertising |
| `<hgroup>` | groups a set of **`<h1>-<h6>`** elements |
| `<header>` | contains the header elements of a section or page |
| `<time>` | represents dates and times within the HTML content |

# User Testing: Google Lighthouse

Lighthouse is an **"open-source, automated tool for improving the quality of web pages"**

- Audits your webpages against a range of criteria
  1. Performance
  2. Accessibility
  3. Best practices
  4. SEO (Search Engine Optimisation)
  5. Progressive web app

- Can be run **within the browser** or as a **Node package from the terminal**

- Try it yourself

# User Testing

**There is always another app – understand your user…**

- What type of experience?
- What is their professional background?
- What are their needs and interests?
- How are they currently trying to meet these needs?
- Where and when would they use this app?

# User Testing: Customer information

## Educated assumptions

- Avoid obvious misconceptions

## Focus groups, interviews…

## Organising customer information

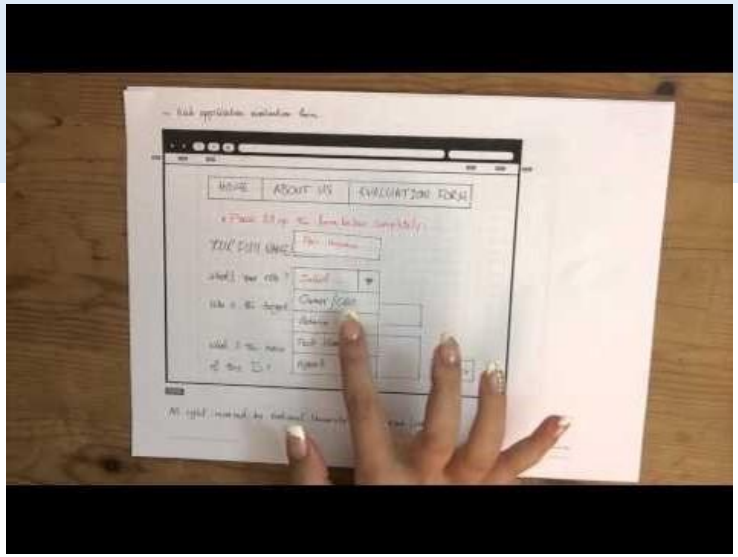
- Assumption personas – description of target users
  - Create the product for someone we believe exists rather than our idea of the user
- Help team members share understanding of audience groups
- Prioritise features by how well they meet user users

# User Testing: User centred design

- **Needs**, **wants** and **limitations** are the focus for the design process

## Five steps for UCD

1. Identify primary features based on personas
2. Understand why the persona needs the features
3. Investigate how other apps provide such features
4. Sketch/wireframe your idea
5. **Test design with users**



# User Testing: Test design with users

## Test with real content

### Three simple questions

- What does this feature do?
- What do you like about it?
- What don’t you like about it?

### Tips for usability testing

- Test the feature, not the user
- Remain neutral – **listen and learn** and don’t assist the user (**failure is informative**)
- Take good notes (audio and/or video record, but only if user consents)
- Keep testing throughout the design process

# Summary

## Unit testing

- Check that your code runs at the function/object level
- Automated code testing
- Test business logic
- Use assertions to confirm that the program state is as it should be
- Use mock objects to replicate parts of the system where necessary

## Integration testing

- Ensure that the components of the system work together

## Accessibility

- Users have a wide range of needs – you must cater for them
- Assess accessibility with Lighthouse

## Usability studies

- Check that the software is usable by your users